Analyzing Properties and Behavior of Service Discovery Protocols using an Architecture-based Approach

Christopher Dabrowski and Kevin Mills
National Institute of Standards and Technology
Gaithersburg, Maryland 20899 USA

Abstract. Current trends suggest that future software components will need to discover and adapt to dynamic changes in available software services and network connections. This implies that future systems may appear as collections of components that combine and recombine dynamically in reaction to changing conditions. Such environments demand new analysis approaches and tools for software design, implementation, and testing. Our work considers how one might rigorously assess the robustness of distributed software systems in response to dynamic change, such as process, node, and link failures. More particularly we seek techniques that can be applied early in the development process to test the behavior and resilience of dynamic distributed systems, and to compare and contrast various approaches to design such systems. As a challenging application we investigate service discovery protocols. We adopt an architecture-based approach that entails the following general steps: (1) construct an architectural model of each discovery protocol, (2) identify and specify relevant consistency conditions that each model should satisfy, (3) define appropriate metrics for comparing the behavior of each model, (4) construct interesting scenarios to exercise the models and to probe for violations of consistency conditions, and (5) compare the results from executing similar scenarios against each model. We elaborate our approach, using Jini as a specific example, and show how Jini can be analyzed using Rapide, an Architecture Description Language (ADL). Our analyses take two forms: property analysis and event analysis. Both depend upon Rapide's ability to execute a specification and to generate events. We use property analysis to investigate robustness to dynamic change, while we use event analysis to discern underlying causes of observed behavior and performance. We argue that static, natural-language specifications largely miss collective behavior arising when various components interact together in a distributed system. We show that a single architectural model can be used to understand both logical and performance properties of a distributed system design. We evaluate how well Rapide supported our modeling and analyses. We also recommend improvements in ADLs to help test and analyze designs for distributed systems.

1. Introduction

Numerous trends suggest that future software will operate in an environment much more uncertain than today's typical client-server paradigm. Increased deployment of wireless communications, implying greater user mobility, coupled with proliferation of personal digital assistants and other information appliances, foretell a future where software components can never be quite sure about the network connectivity available, about the other software services and components nearby, or about the state of the network neighborhood a few minutes in the future. In the most extreme situations, as found for example in military applications [1], software components composing a distributed system may find that cooperating components disappear due to physical or cyber attacks or due to jamming of communication channels or movement of computing platforms beyond communications range. Even in less demanding circumstances, increased use of computer chips, network communications, and software to implement a growing range of consumer appliances portends the need for simple, self-contained units that, when powered on, can discover their technical surroundings and then automatically configure themselves into a larger system that might already be deployed. Further, as the consumer rearranges components in such a system, then the system must automatically adapt its configuration as necessary. Such environments demand new analysis approaches and tools for software design, implementation, and testing.

Our work considers how one might rigorously assess the robustness of distributed software systems in response to dynamic change, such as process, node, and link failures of both a temporary and permanent nature. More particularly we seek techniques to test the behavior and resilience of dynamic distributed systems, and to compare and contrast various approaches to design such systems. As a challenging application we investigate service discovery protocols, which provide mechanisms for rendezvous and robustness in the face of uncertainty. Such mechanisms enable dynamic elements in a network: (1) to discover each other, (2) to express opportunities for collaboration, and (3) to compose themselves into larger collections that cooperate to meet an application need. In this paper, we limit our analysis to Jini(tm)[1] Networking Technology, one of at least six service discovery protocols [2-7] designed to date. Future papers will consider additional discovery protocols.

We wish to address software robustness as early as possible in the engineering lifecycle because the earlier a design error can be uncovered, the lower the cost to repair. For this reason, we use an Architectural Description Language (ADL) [12-19] to transform natural-language specifications into architectural models that provide rigorous representation of system structure and behavior. Such architectural models, coupled with appropriate automated analysis tools, permit designers to uncover and correct errors and omissions, and to clarify ambiguities that would otherwise lead to incorrect behavior, or to performance problems, after a specification has been implemented and the resulting software deployed. Architectural models also provide significant advantages over less formal approaches when comparing and contrasting alternate designs for dynamic distributed systems, such as service discovery protocols.

Other authors compare various service discovery protocols [8-11, 22, 24]. While instructive, these comparisons exhibit significant limitations. For example, existing comparisons are largely functional in nature and informal in presentation. Such comparisons cannot capture nor express a deep understanding of the behavioral properties of the protocols, nor can these comparisons uncover areas of ambiguity, inconsistency, and incompleteness within the specifications. Further, existing comparisons use concepts and terminology taken from individual specifications. Since each specification adopts a unique language for describing its design, it becomes difficult to compare the designs directly. In future work, we aim to contribute a more rigorous comparison of three discovery protocols: Jini [4], UPnP [3], and SLP [6].

The current study serves two purposes: (1) validate our approach against the specification for Jini and (2) evaluate the suitability of ADLs to model and analyze dynamic distributed systems. To perform this study, we examined several ADLs [12-19], selecting Rapide [12], an ADL developed at Stanford

[1] Certain commercial products or company names are identified in this report to describe our study adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the products or names identified are necessarily the best available for the purpose.

University. Rapide specializes in modeling architectures for real-time, distributed systems and therefore represents behavior in a form suitable to investigate discovery protocols. Rapide also comes with an accompanying suite of analysis tools that can execute a specification and can record and visualize system behavior.

This paper reports our initial results with respect to modeling and analyzing the Jini specification. The paper is organized as five sections. First, we describe our approach to model and analyze discovery protocols. We provide a general architecture intended to encompass all the protocols we studied. Using Jini as an example, we illustrate how this architecture can be used to model a specific protocol, and then how the model can be converted to an executable specification, described using Rapide. In the second section, Analysis Approaches, we discuss the application of ADL tools to analyze logical properties of our models, and in the process to uncover specification deficiencies, and to assess the degree to which the model satisfies selected consistency conditions. Further, we show how behavior traces from our model can be analyzed to produce quantitative metrics. In the third section, we report and discuss the results obtained from our initial analysis of Jini. We examine how well our Jini model satisfies selected consistency conditions, and we characterize the behavior and performance of Jini with respect to particular scenarios. In the fourth section, we assess our experiences using an ADL and related tools to model and analyze Jini. We report our positive findings, along with recommendations for improvements. In the fifth section, we provide our conclusions and outline future work.

2. Modeling Discovery Protocols with an Architecture-based Approach

Most extant discovery protocols are specified statically, using natural language, and supplemented with reference software that provides one presumably legitimate implementation of the specification. The static specification expresses the appropriate behavior of system components in reaction to particular events and conditions. The reference implementation contains incidental complexity needed to fit the protocol into a software framework that includes various supporting components. Typically, static specifications cannot be used effectively to understand the dynamic behavior of distributed systems. Such specifications do not express collective behavior very well and often do not define consistency conditions against which dynamic behavior can be evaluated. Further, natural-language specifications usually lack completeness, and suffer from ambiguities and inconsistencies. On the other hand, reference software includes complexity irrelevant to the fundamental requirements of the specification. Further, reference software typically will implement one particular design choice in cases where a specification may allow various alternatives.

To overcome these shortcomings, we adopted an approach that entails the following general steps: (1) construct an architectural model of each discovery protocol, (2) identify and specify relevant consistency conditions that each model should satisfy, (3) define appropriate metrics for comparing the behavior of each model, (4) construct interesting scenarios to exercise the models and to probe for violations of consistency conditions, and (5) compare the results from executing similar scenarios against each model. Below, we elaborate our approach, using Jini as a specific example, and show how Jini can be modeled using Rapide, an Architecture Description Language (ADL). We also discuss the Rapide run-time, which converts our Jini model to an executable specification. First, we introduce discovery protocols, and define some consistent terminology that we can use to build comparable architectural models.

2.1 Discovery Protocols in Essence. Discovery protocols enable software components to find each other on a network, and to determine if discovered components match their requirements. Further, discovery protocols include techniques to detect changes in component availability, and to maintain, within some time bounds, a consistent view of components in a network. Many diverse industry activities explore different approaches to meet such requirements; leading to a variety of proposed designs for service discovery protocols [2-7]. Some industry groups approach the problem from a vertically integrated perspective, coupled with a narrow application focus. Other industry groups propose more widely applicable solutions. For example, a team of researchers and engineers at Sun designed a general service discovery mechanism atop Java(tm), which provides a base of portable software technology. The proliferation of service discovery protocols motivates deeper analyses of their designs.

Beyond this, given the level of debate within the industry, a comparative analysis can help to assess the relative merits of particular protocols.

Table 1. Mapping Concepts Among Discovery Protocols

Generic Model	Jini	UPnP	SLP
Service User	Client	Control Point	User Agent
Service Manager	Service or Device Proxy	Root Device	Service Agent
Service Provider	Service	Device or Service	Service
Service Description	Service Item	Device/Service Description	Service Registration
Identity	Service ID	Universal Unique ID	Service URL
Type	Service Type	Device/Service Type	Service Type
Attributes	Attribute Set	Device/Service Schema	Service Attributes
User Interface	Service Applet	Presentation URL	Template URL
Program Interface	Service Proxy	Control/Event URL	Template URL
Service Cache Manger	Lookup Service	not applicable	Directory Service Agent (optional)

To help us compare protocols, we developed a general UML (Unified Modeling Language) model, expressed with a consistent terminology that provided a basis for the Rapide architectural model. The main components in our general model include: (1) service manager (SM), (2) service user (SU), and (3) service cache manager (SCM), where the service cache manager is an optional element not supported by all discovery protocols. These components participate in the discovery, registration, and consistency maintenance processes that comprise dynamic discovery protocols. A service manager maintains a database (Service Repository) of records (Service Descriptions, or SDs), where each record describes the essential characteristics of a particular service or device (Service Provider, or SP). Each SD contains the identity, type, and attributes that characterize a SP. Each SD also provides up to two interfaces (an application-programming interface and a graphic-user interface) to access a service. Table 1 shows how these general concepts map to specific concepts for Jini, UPnP, and SLP. Since the paper uses Jini as an example, we provide a brief synopsis.

2.2 Jini in Brief. Upon startup, a Jini component (SU, SM, or SCM) engages in a discovery process to locate other, relevant Jini components within the network neighborhood. To oversimplify things: (1) SMs attempt to discover relevant SCMs with which to register a SD for each SP managed and (2) SUs attempt to discover relevant SCMs to query for SDs that lead to desired SPs. In other words, SUs and SPs rendezvous through SDs registered by SMs with particular SCMs, where the SCMs are found through a discovery process.

2.2.1 Jini Discovery. Jini encompasses two discovery modes, *multicast and directed*, supported by three discovery processes, which we call aggressive, lazy, and directed. Both aggressive and lazy discovery involve multicast communication among Jini components participating in two multicast groups. Upon initiation, a Jini component enters aggressive discovery, where it transmits probes at a fixed interval for a specified period, or until it has discovered a sufficient number of SCMs. Upon cessation of aggressive discovery, a component enters lazy discovery, where it listens for announcements sent at intervals by SCMs. This implies that during lazy discovery a SCM both listens for announcements by other SCMs and sends its own announcements at the required intervals. Figure 1 gives a simplified illustration of the two Jini discovery modes, and the three supporting processes.

During aggressive discovery, probes sent by Jini components identify interest in one or more administrative scopes, which Jini calls groups; probes also contain a list of SCMs already discovered by the Jini component. Each SCM must reply to a probe only when the list of groups contained within the probe intersects with the SCM's own list of groups in which it is a member, and also provided that the probe does not indicate that the SCM has already been discovered. Once a relevant SCM is discovered, the discovering component requests an application-programming interface (API) that enables the component to interact with the SCM.

Lazy discovery operates similarly. Announcements sent by SCMs identity group membership. A Jini component requests an API from an announcing SCM when the following conditions hold: (1) the group membership of the SCM intersects with the groups of interest to the component, (2) the component has not already discovered the SCM, and (3) the component has not already discovered enough SCMs. Receipt of an API from the SCM ends the discovery process between the component and the SCM.

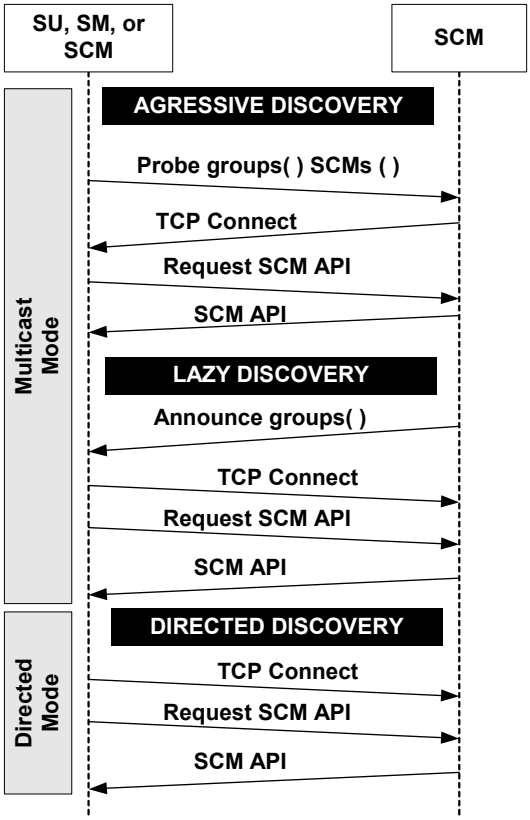

Figure 1. The Three Jini Discovery Processes

Directed discovery operates differently from multicast discovery. Each Jini component may be given a specific list of SCMs to discover. For each SCM on the list, a Jini component establishes a connection and requests an API. Should the SCM prove unavailable, the component can continue to retry connecting. As explained later, ambiguities regarding interaction between directed and multicast discovery lead to several problems for the Jini specification.

Once a Jini component obtains an API from a SCM, the component can use the API to access services provided by the SCM. To allow the component and the SCM to reside on different network nodes, the API must use a communication protocol, such as Java Remote Method Invocation (RMI)[2], which enables the component to access SCM services as if they resided within the same Java Virtual Machine (JVM). In general, SCM services can be classified as registration and consistency maintenance, which Jini refers to as leasing.

2.2.2 Jini Registration. A SM holds a SD for one or more SPs. The SM must register each of these SDs with each SCM discovered. As part of the registration request, the SM asks that the registration remain valid for some duration. If the SCM agrees to add the SD to its set of registered services, then the SCM grants a lease time (not more than requested) and returns a service item and lease to the SM. Once a SD is registered with a SCM, SUs can discover the existence of the related SP by querying the SCM, or by receiving notifications from the SCM. Before receiving notifications, a SU must register notification requests with a SCM. A SU can register a request that a SCM notify the SU whenever the SCM adds, deletes, or changes a SD of interest. As with service registrations, notification requests will be maintained by a SCM only for an agreed time (the lease period).

2.2.3 Jini Consistency Maintenance. In a distributed system, new services and devices can be deployed, obsolete services and devices can be removed, and nodes, processes, and links can fail. These facts imply that replicated state, distributed throughout a system, can become inconsistent. To time bound such inconsistencies, Jini requires each SCM to periodically purge SD registrations and notification requests. For this reason, a SCM assigns a lease to each registration and notification request. The lease indicates when the SCM plans to purge the item. To prevent its removal by the SCM, the registering component must renew the lease prior to the purge time. In this way, if the registering component fails (or the network path fails), then the SCM can, within a bounded delay, remove reference to the item, and, when appropriate, can notify other interested components. Once the failure is resolved, the discovery and registration processes can be restarted for the failed component, and the previous state might be recovered eventually.

Interactions with SCMs provide another means for Jini to maintain consistent state. Each component may register some items with a SCM. In addition, leases for these registered items must be renewed periodically. Whenever a component attempts to invoke a SCM method across a network the possibility exists for a remote exception. Remote exceptions indicate that the corresponding SCM (process or node) might have failed, or that the network link between the component and the SCM might have failed. A component is free to retry a method invocation, and to give up after some period of time.

[2] Jini does not require the use of any particular technique for remote procedure calls. In this paper, we use RMI for illustrative purposes.

2.3 Complexity and Uncertainty. The foregoing discussion of Jini, while oversimplified, highlights the inherent complexity and uncertainty associated with discovery protocols. Complexity arises from several sources. The protocol involves multiple parties communicating across a network, which introduces asynchrony, and which can also introduce variable delays. Multiparty interactions can be quite difficult to specify and understand. Further, the protocol defines various operating modes that could potentially interfere with one another, and each protocol entity maintains independently operating behavioral threads, which implement features that can interact in unanticipated ways.

Uncertainty also arises because nodes, processes, and links can appear and disappear without warning. Discovery protocols must include behavior to cope with such changes. The coping behavior itself can exhibit unexpected interactions with the already complex behavior defined to implement multiparty communication. Together, this complexity and uncertainty discourage protocol designers from attempting to specify the properties of a particular discovery protocol. Yet, we desire to compare and contrast the protocols based on such properties. This conundrum led us to the idea of constructing an architectural model for each discovery protocol, and using the models to investigate various properties.

2.4 Elements of an Architectural Model. Broadly speaking, an architectural model comprises a set of components, and the connections among them, along with the relationships and interactions among the components. In our application, an architectural model expresses structure (as components, connections, and relations), interfaces (as messages received by components), behavior (as actions taken in response to messages received, including generation of new messages), and consistency conditions (as Boolean relations among state variables maintained across different components).

2.5 An Architectural Model for Jini. Figure 2 depicts the top level of our Jini architecture that was realized in Rapide. This architecture consists of three component types (SU, SM, and SCM) together with three connection types: Aggressive Discovery Multicast Group (ADMG), Lazy Discovery Multicast Group (LDMG), and Remote Method Invocation Unicast Link (RMIUL). Only one instance each can exist for the LDMG and ADMG but the SU, SM, SCM, and RMIUL can be instantiated as multiple instances. Each SU, SM, and SCM resides on a network node and participates in service discovery, registration, and consistency maintenance. To perform these functions, each type of Jini component is decomposed into subcomponents (not described in this paper due to lack of space). Jini components use the ADMG to distribute probes to any SCMs listening. SCMs use the LDMG to distribute announcements to any Jini component listening. When asked to engage in directed discovery, a Jini component uses one RMIUL to contact each SCM on its directed-discovery list. To invoke methods on a specific SCM, a Jini component must use an appropriate RMIUL.

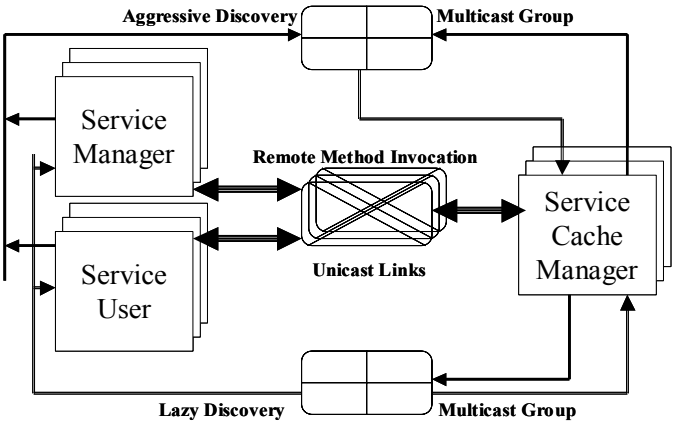

Figure 2. Top-Level Architectural Model of Jini

We implement SMs, SCMs, and SUs, as Rapide interfaces. We define connections, also implemented as Rapide interfaces, to link Jini components that exchange events. We use Rapide services to constrain the event types allowed on each connection. We model two classes of connection: (1) fan-out multicast links (ADMG and LDMG) for discovery and (2) unicast links (RMIUL) for directed discovery and for remote-method invocation.

Modeling connections as Rapide interfaces allows the links to encapsulate logic: (1) to control link state (up or down) and (2) to send appropriate remote exceptions in response to events sent over a failed link. The remote exception logic proves significant because some events require remote exceptions to be sent in one direction, while other events require bi-directional remote exceptions. Since nodes may come up or go down at any time, our model also includes specific events to start and stop nodes. As we discuss later (Section 5.3.2) these requirements have implications for how ADLs should model connections.

3. Analysis Approach

Our specification analyses take two forms: property analysis and event analysis. Both depend upon Rapide's ability to execute a specification and to generate events. We use property analysis to investigate robustness to dynamic change, including network failure. Property analysis also provides insight into processes defined in a protocol specification, and helps to identify ambiguity, inconsistency, incompleteness, and other flaws.

Event analysis examines *Rapide* POSETs (partially ordered sets of events exchanged among components) to discern underlying causes of observed behavior and performance, and especially to assess the protocol's capacity to recover from network disruption. We also use event analysis to understand circumstances surrounding specific protocol design issues, such as race conditions.

Property and event analysis can be used together to evaluate a protocol's resilience in the face of network failure. We also suspect that POSETs can provide a basis for complexity metrics, another dimension along which we expect to compare discovery protocols. Our current work has not developed such complexity metrics. Below, we describe our use of *Rapide* to analyze properties and behavior of Jini.

3.1 Property Analysis. To implement property analysis we define consistency conditions and then use the Rapide constraint language to express the negation of each consistency condition. If a negation is satisfied, then Rapide has detected an inconsistency. We stimulate periodic events, called *consistency probes*, which retrieve values from the internal state variables of appropriate components. At each probe interval Rapide checks for the presence of an inconsistency. In general, discovery protocols attempt to guarantee time-bounded inconsistency. Our analysis strives to verify such guarantees. We also seek to identify unbounded inconsistencies, which persist indefinitely. Unbounded inconsistencies suggest areas of a specification, or protocol design, which merit further attention. Below we give some examples of consistency conditions. In Section 4, we discuss circumstances in our Jini model where these consistency conditions do not hold.

3.1.1 Sources for Consistency Conditions. We posited the quality of service that users might expect from discovery protocols. Then we defined these ideas as consistency conditions that specify relationships a protocol should strive to maintain among state variables across interacting components. In this paper, we define selected consistency conditions[3] that should hold in the absence of failures or other dynamic changes that could permit the conditions to be violated for a transient period.

3.1.2 Consistency among Distributed Components. Several consistency conditions concern the SCM and the SM. Analogous conditions could also be defined for the SCM and the SU. For example, a SM can only register a service description with a SCM it has discovered. This can be expressed as the following consistency condition:

For All (SM, SD, SCM):
(SM, SD) IsElementOf SCM registered-services (CC1)
implies SCM IsElementOf SM discovered-SCMs

In our model, we express the negation of this consistency condition as a Rapide constraint. Consistency probes return the contents of each SM's list of discovered SCMs and of each SCM's list of registered services. Rapide checks various combinations of values for specific pairs of SMs and SCMs at each probe time. When the negation is true, an inconsistency exists.

A second example consistency condition states that if a SM has discovered a SCM and the SM has a SD for a service that it is managing, then the SM should have registered the SD with the SCM. Here, a service is managed if the SM is required to advertise its availability. This may be expressed as:

For All (SM, SD, SCM):
SCM IsElementOf SM discovered-SCMs & (CC2)
(SD) IsElementOf SM managed-services
implies (SM, SD) IsElementOf SCM registered-services

[3] Consistency conditions we define here do not necessarily reflect the intent of Jini's designers.

This consistency condition amounts to an inverse view of CC1. This inverse view can catch specification issues that CC1 would miss.

A third example consistency condition states that if a SM has discovered a SCM through multicast discovery and has registered its services on that SCM, then there should be an intersection between the list of groups the SM is to join and at least one group in which the SCM holds membership. This can be expressed as:

For All (SM, SD, SCM):
SCM IsElementOf SM discovered-SCMs & (CC3)
(SM, SD) IsElementOf SCM registered-services &
NOT (SCM IsElementOf SM persistent-list)
implies Intersection (SM GroupsToJoin, SCM GroupsMemberOf)

Reference to the absence of membership of the SCM in the SM persistent list eliminates SCMs that the SM found through directed discovery.

3.2 Event analysis. We use event analysis to understand underlying causes for the observed behavior and performance of discovery protocols. The general idea is to define a set of usage scenarios that can be executed against the models of several discovery protocols. Table 2 provides an excerpt from a scenario we defined, and provides a sense of the stimuli that can be simulated. While executing scenarios, the Rapide run-time produces POSETs that provide a basis for analyses. POSETs help us to understand relationships among events, which trace back to specific behavior in components, and to possible issues within a specification. The POSETs may also be used to compute simple metrics, such as number of events generated or time taken by the model to transition between two configurations of interest. To support such computation, we insert performance probes at key points in the Rapide model. Such probes can compute the desired measurements, or can place markers in the POSET for off-line computation. While event analyses can be applied individually to specific protocols, greater value may accrue in comparative analysis. Following we give examples of some event analyses of interest.

Table 2. Scenario Excerpt

Time	Command	Parameters
5	NodeFail	SM4
5	LinkFail	SCM1 SM4
10	GroupJoin	SM4 GROUP1
10	FindService	SU8 5 1 2 S XYZ ALL
50	AddService	SM4 SCM3 T ATT API GUI 20 30

3.2.1 Identifying and Understanding Race Conditions. Due to asynchronous processing and associated delays in communications among components, distributed systems often exhibit race conditions, where system behavior can vary depending upon the order in which events arrive at cooperating components. Though such problems cannot always be eliminated, it remains important to identify the existence of specific race conditions so that application programmers can adopt appropriate safeguards. We can use Rapide to find race conditions by asserting and testing consistency conditions. For example, consider the following:

For All (SM, SD, SCM, SU, NR):
(SU, NR) IsElementOf SCM requested-notifications & (CC4)
(SM, SD) IsElementOf SCM registered-services &
Matches((SM, SD), (SU,NR))
implies (SM, SD) IsElementOf SU matched-services

This consistency condition indicates that if a SU has requested notification when a certain service (SM, SD) registered at a SCM matches specified criteria, then the SU should become aware of the matching service. While the Jini specification does not guarantee CC4, we would be interested to identify situations where the condition does not hold. In such cases, we can analyze the POSET to determine specific causes. In this way, we might uncover race conditions that require an application programmer to take particular care when using Jini's matching mechanisms.

*3.2.2 Measuring and Understanding Protocol Performance***.** When comparing various discovery protocols, we can use Rapide to define and compute performance metrics, and then use POSET analysis to investigate the underlying behaviors. Of course, comparative performance must be considered in light of selected scenarios of interest. For example, consider a scenario where a major power failure occurs after the discovery phase has completed, services and notification requests are registered, and SUs have received SDs for services that meet their requirements. During the failure, most Jini entities lose some internal state: all nodes lose discovered SCMs; SUs lose SDs for services previously discovered; but SCMs and SMs must retain specified persistent information. Upon power restoration, the Jini components restart and recover. To assess recovery performance we define two metrics, restoration latency and restoration overhead, which measure the efficiency of recovery in terms of total time and number of messages generated before all SUs rediscover their original set of SDs. Restoration latency covers node start-up delays, transmission times, processor background workload, and times for processing transaction data. Restoration overhead includes all events exchanged by Jini components from power up through complete restoration of the desired state.

4. Selected Analyses of the Jini Service Discovery Protocol

In this section we discuss some results obtained running scenarios against our Jini architectural model. We were able to verify the robustness of Jini's design in a range of failure scenarios that are not presented here due to lack of space. However, we found the Jini specification unclear regarding interactions between multicast and directed discovery. In particular, we could not discern whether discovered SCMs should be kept on a single list or whether SCMs found by directed discovery should be kept on a separate list from SCMs found by multicast discovery. We included both interpretations in our Jini model, and we ran related scenarios to evaluate CC1 and CC2. We also noticed that the Jini reference implementation permits administrators to alter the operation of a running SCM. We were interested to consider if such changes could adversely affect a Jini network, so we ran related scenarios to evaluate CC3. Further, we discovered an apparent race condition that is difficult to discern from reading the Jini specification, so we ran scenarios to evaluate CC4. We also executed selected scenarios to understand some performance characteristics of Jini systems. Here, we discuss restart from power failure. While the work described here suggests some incompleteness and ambiguity (already shared with Sun) in the Jini specification, our purpose is to illustrate an architecture-based approach to model, analyze, and compare service discovery protocols. Regardless of any ambiguity and incompleteness discussed here, overall we found Jini to operate as specified.

4.1 Interfering Interactions between Directed and Multicast Discovery. The Jini specification permits a Jini component to engage simultaneously in two modes of discovery: directed and multicast. However, the specification is unclear with respect to issues that arise regarding interactions between these two modes. This means that an implementer must make some decisions, which can lead to various difficulties. We identified decisions that cause local interference between independent processes on the same Jini node. We also found decisions that cause independent processes on the same Jini node to interfere with the node's remote state on discovered SCMs. We discuss these situations below.

4.1.1 Local Interference. For the following discussion, assume that the implementer decides to maintain a single list of SCMs discovered by a SM. Figure 3 illustrates (using a simplified description) what occurs during a scenario where SM4 uses multicast discovery to find SCMs in a Jini group (GROUP2). In this case SM4 discovers SCM3, also a member of GROUP2. Shortly after, SM4 is told to discover SCM3 (AddSCM) through directed discovery, and at the same time SM4 is told to drop membership in GROUP2. The resulting behavior leads to a time-bounded violation of CC1, which states that a SD should not be registered on a SCM if the SCM is not on the discovered list of the SM managing the SD. The specific behavior follows.

Through multicast discovery SM4 finds SCM3 and adds it to the list of discovered SCMs. Subsequently, SM4 is asked simultaneously to leave GROUP2 and to discover SCM3. The group leave causes SM4 to first cancel leases for SDs held on SCM3 and then to remove SCM3 from its list of discovered SCMs. Between these two events, SM4 uses directed discovery to find SCM3 and then

attempts to add SCM3 to its list of discovered SCMs. Since our model assumes that probes will be built from the list of discovered SCMs, we decided not to insert duplicate SCMs in that list.[4] This rule is enforced by the list maintenance function. Therefore, in Figure 3, the second discovery of SCM3 is not added to the list of discovered SCMs because it's already there. Soon thereafter, SM4 completes lease cancellation for SDs on SCM3 and then removes SCM3 from its list of discovered SCMs. In the meantime, the directed discovery process in SM4 registers SDs with SCM3. At that point, CC1 is violated, and remains so until the leases for the SM4 SDs expire on SCM3.

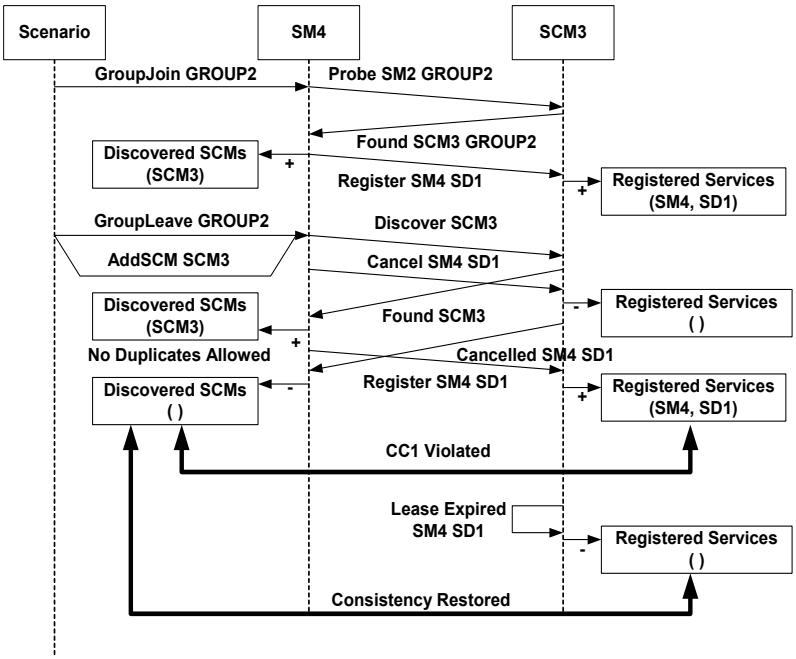

Figure 3. Local Interference between Directed and Multicast

4.1.2 Remote Interference. Suppose that an implementer decides to maintain SCMs discovered by multicast and directed discovery on separate lists? In this case, local interference disappears, only to be replaced by a form of remote interference, where two discovery processes within the same node independently manipulate the state of SDs on SCMs. Figure 4 illustrates behavior from a scenario that uncovers this problem through violation of CC2, which states that services managed by a SM must be registered on all discovered SCMs. In the scenario, SM4 uses directed discovery to find SCM1. Later SM4 is instructed to join GROUP1, which includes SCM1. This causes a duplicate service registration, which leads SCM1 to abrogate the existing lease for (SM4, SD1). Subsequently, SM4 is told to leave GROUP1. In the end, this causes SM4 to cancel leases for its SDs held on SCM1, resulting in a situation where SCM1 is on the list of SCMs discovered directly by SM4 but where the SDs from SM4, which were originally registered through the directed discovery action, are not now registered on SCM1. Assuming that SM4 maintains a single registration process, this violation of CC2 is unbounded in time.

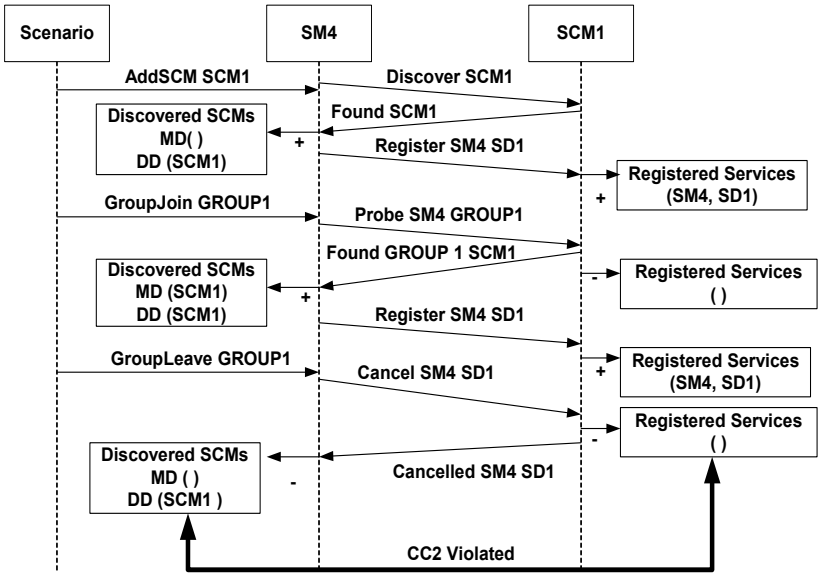

Figure 4. Remote Interference between Directed and Multicast Discovery

4.2 Insensitivity to Changes in Group Membership by SCMs. The Jini reference implementation includes an interface that permits an administrator to alter parameters associated with a running SCM. We

[4] Allowing duplicates on a single list leads to a number of other problems, which are beyond the scope of the discussion here.

mirrored this behavior within our Jini model, and then exercised the option to change group membership of a running SCM. Figure 5 illustrates the relevant subset of a related scenario. First, SM4 is instructed to join GROUP1, which leads to the multicast discovery of SCM1 (a member of GROUP1). Subsequently, an administrator removes (AdminDeleteGroup) SCM1 from membership in GROUP1. Once this occurs, CC3 is violated because: (1) SM4 has found SCM1 with multicast discovery, (2) SDs managed by SM4 are registered with SCM1, and yet (3) SM4 and SCM1 have no common group membership. The violation of CC3 continues in a time-unbounded form so long as SM4 renews leases on SCM1.

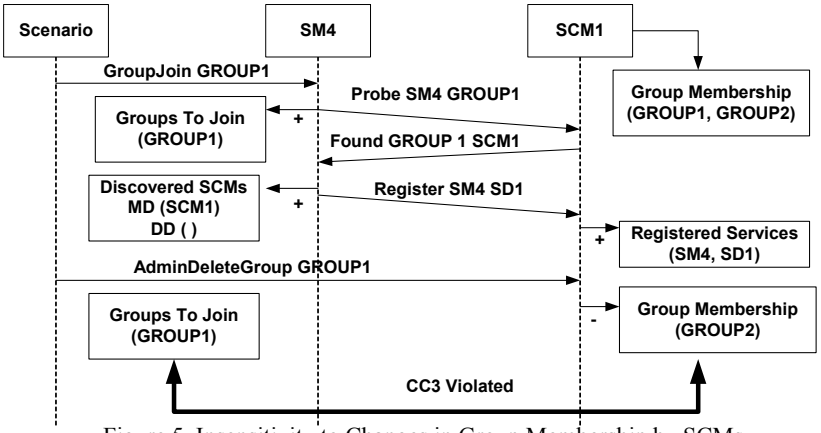

Figure 5. Insensitivity to Changes in Group Membership by SCMs

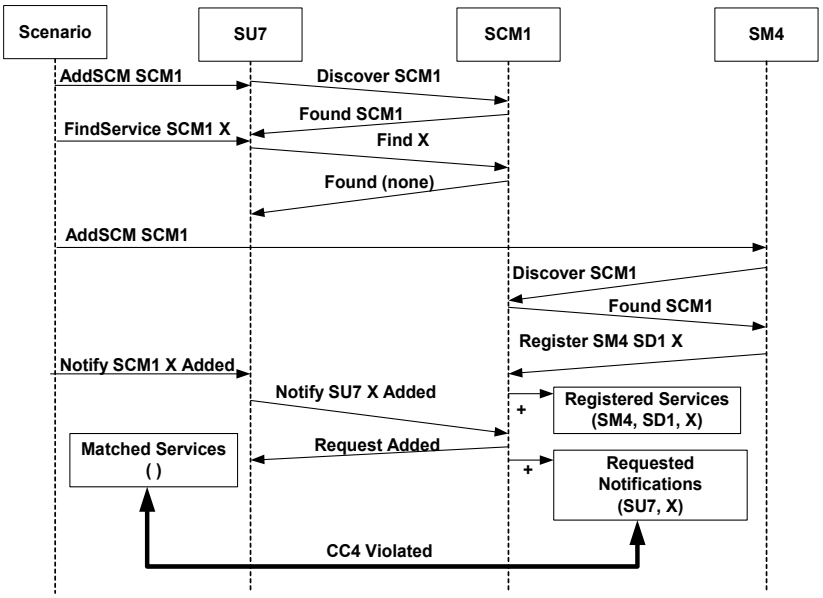

Figure 6. A Sample Registration Race Condition

These results suggest that the Jini specification may be incomplete with regard to this issue. While an administrator can remove group membership from a running SCM, the Jini protocol specifies no behavior in reaction to this new information. As a SCM continues to issue announcements, which contain its current group membership, other Jini components are told to ignore announcements from SCMs that do not belong to groups of interest. As shown in the discussion above, this can lead to a situation where SMs (as well as SUs) may continue to maintain registration with SCMs no longer relevant. This might or might not be the intent of Jini's designers; however, the issue should be addressed in the specification.

4.3 Race Conditions. All distributed systems exhibit the possibility for race conditions. Our architectural model permits us to investigate how such conditions can arise. Figure 6 presents a portion of a scenario illustrating a race between service registration by SMs and registration of notification requests by SUs. In this case, SU7 discovers SCM1 and then queries it for a matching service. At the time of the query, SCM1 does not contain a SD for a matching service and so replies without matches. In this particular scenario, SU7 delays for 10s its request to be notified by SCM1 when a SD for a matching service is added to the SCM cache. In the interim, SM4 discovers SCM1 and registers a SD for a service matching the needs of SU7. Unfortunately, the only matching service was registered during the interval between the query and the request for notification by SU7. In Jini's definition of matching semantics, SU7 can continue to renew leases for its request for notification and SM4 can continue to renew leases for its SD and the two will never learn of each other. This situation results in a time-unbounded violation of

CC4, which states that if a SCM holds a notification request from a SU, which matches a SD also held by the SCM, then the SU should know about the matching SD.

While this violation of CC4 can be attributed to the 10s delay before SU7 sends a notification request, a number of other situations can lead to similar results. For example, network congestion can delay the reply to the original query by SU7 or can delay the request by SU7 for SCM1 to register its notification. Alternatively, competing processing within the node supporting SU7 could delay the generation of its notification requests. To account for this, SUs might issue a second query for a matching service after the notification request is registered with a SCM. In this way, the SU can detect any matching SDs registered by the SCM after the first query but before the notification request.

4.4 Restart Performance. To demonstrate the ability of our architectural model to provide insight into performance-related behavior, we describe the results of an experiment to investigate the restart of a Jini network following recovery from a major power failure. The experiment topology consists of nine nodes (three of each type: SU, SM, and SCM). We partition the nodes into threes, where each partition consists of one SU attempting to rendezvous with one SM through a SCM. Once all SUs have found their assigned SMs, we simulate a major power failure, which causes all nodes to crash for 40s. We then restore the power and wait for all SUs to rendezvous with their assigned SMs. Table 3 gives the values for relevant experiment parameters. Upon restart, each Jini node chooses a random delay before beginning discovery; we used delays uniformly distributed between two and 15s. We also had each SU and SM request leases of 30s for notification requests and service registrations, and we had each node renew the leases for a period of 100s. For each link, we introduced variable transmission delays; for each node, we introduced variable processing-load delays. We also introduced processing delays for manipulating items in the discovery databases and the SCM registration databases. Since the Jini specification did not address the persistence of notification requests upon SCM failure, we assumed that this information was purged on failure.

Table 3. Parameters for Power Failure Restart Experiment

	Parameter	Value
Jini Parameters	Node Restart Delay	2s – 15s uniform
	Probe Interval (and period)	5s (7 times)
	Announce Interval	120s
	Per Lease Time	30s
	Total Leasing Duration	100s
	Notification Requests	Purge on SCM Failure
Transmission and Processing Delays	Transmission Delay	1us – 10us uniform
	Processing Load Delay	10us – 100 us uniform
	Per Item Processing	10 us (discovery DBs) 100 us (SCM cache)

We ran the experiment 30 times, measuring the restoration latency and overhead. In this experiment, before the original state could be recovered, all nodes had to restart. For that reason, the maximum node restart delay dominates the restoration delay. For example, for our experiment runs, the average maximum node restart delay was 12.56s (2.09s variance), and the average restoration latency was 14.76s (3.31s variance). The restoration overhead in each run depends upon the restoration latency, because periodic message exchanges associated with Jini discovery and leasing continue through the restoration. In this experiment, the restoration latencies were relatively close, as were the number of messages exchanged, differing only in the number of probes sent during aggressive discovery and in the subsequent number of discoveries. In our runs, the number of messages exchanged to achieve restoration ranged approximately between 70 and 90. These results demonstrate that the same architectural model can be used to investigate both performance and logical properties of a distributed system.

4.5 Summary of Findings. Using our architectural model and usage scenarios we were able to verify the robustness of Jini mechanisms in a range of failure scenarios. Further, as supported by the analyses above, we were able to uncover areas of incompleteness and ambiguity in the natural-language specification for Jini. While a static, natural-language specification, such as Jini's, contains a reasonable description of the behavior of each component in response to specific events, such specifications largely miss collective behavior arising when various components interact together in a distributed system, and especially when pieces of the system change state during the interactions. In addition, our dynamic,

executable model of the Jini specification permitted us to explore the behavior and performance of Jini systems in various realistic scenarios. A static specification cannot hope to provide similar insights.

5. Assessment of the Architecture-based Approach

As part of our work, we assessed how well the Rapide ADL and analysis tools supported our modeling and analyses of Jini, with specific attention to analysis of dynamic behavior. We found that the Rapide ADL provided valid abstractions to represent and analyze the structure and behavior of Jini under conditions of dynamic change. Using Rapide interfaces we easily represented the major service discovery components, and subcomponents (not discussed in this paper). The components proved easy to connect into architectures that model a network of Jini entities. Our analyses relied upon Rapide's ability to represent dynamic behavior through events, rules, and constraints, and then to analyze the resulting POSETs. The ability to represent the behavior of individual components and to analyze the collective behavior resulting from interactions was key, without which this analysis could not have been performed. We did identify some suggestions for improving specific capabilities that apply generally to all ADLs. Before discussing these suggestions, we describe general merits of using an architectural model.

5.2 Merits of Using an Architectural Model. Our Rapide model provided benefits for analysis. Some of these benefits apply to all ADLs. First, the architectural model proved more precise, concise, and informative than the natural-language specification. For example, the architectural model provided executable behavior so that we could discover interactions not apparent from the paper specification. As a consequence, we were able to identify and address areas of ambiguity, inconsistency, and incompleteness. While the Jini specification was supported by a reference implementation, the architectural model proved easier to understand and analyze, and permitted us to focus on the essential complexity inherent in the specification. The reference implementation entailed incidental complexity that interfered with our ability to gain a clear understanding of the behavior of the specification. Second, a single architectural model can be analyzed for behavioral, performance, and logical properties. Using a single model limits the errors and inconsistencies that can creep in when multiple models must be used to represent the same specification. Third, using an architectural model enabled us to readily consider alternative implementation options, where they were allowed by the specification, and to identify specification ambiguities. When addressing ambiguities, the architectural model enabled us to investigate the ramifications of various alternate resolutions.

5.3 Areas for Improvement Below, we identify and discuss some suggestions for improving ADLs in several areas: domain-specificity, simplification through views, representation of structure and behavior, and support for analysis. While we discuss these suggestions in the context of Rapide, we believe they apply more generally to use of ADLs for modeling architectures for dynamic systems.

5.3.1 Need for customizable domain-specific syntax and semantics. Constructing an architectural model typically entails a partnership between a domain expert and a system architect. The partnership proceeds more smoothly when the architecture reflects the terminology of the domain, allowing the domain expert to review the specification with less help from the architect. For this reason, ADLs should support renaming common ADL constructs such as interfaces, components, connectors or modules to use terms familiar in the domain. This would allow the expert to more easily read the specification without having to learn the ADL in detail. The same benefit may accrue from allowing customization of language syntax to be more familiar to domain practitioners, especially with respect to system behavior.

5.3.2 Improvement to representation of structure. Rapide, and other ADLs, connect components to subcomponents and pass events in a strictly hierarchical manner. One purpose in doing this is to constrain communications among subcomponents of different hierarchies in order to limit the introduction of errors when replacing subcomponents. This requires inter-component events to propagate through multiple levels in two hierarchies, leading to several inefficiencies. First, if the same events must be duplicated as a result, an unnecessarily large set of events will be created for analysis. Second, the architecture entails an increased number of connections, resulting in a larger specification, which is more difficult to maintain and modify. This inhibits revision and evolution of system designs, especially important when modeling dynamic systems, and also discourages investigation of alternative design approaches. Third, the strict

hierarchy arrangement does not agree with real-world designs in which subcomponents of different systems often communicate directly. To address these problems, we recommend investigating alternative ways to specify connectivity between top-level components and subcomponents in an architectural model, while preserving correct communications. We plan to address this area further in future work.

Beyond the question of number, connections take on importance for modeling reasons. Specifically, we believe that connections should be represented as first-class entities [17, 20-21, 23]. Many domains, including networking, have numerous, well-known connection classes. Such domain-specific knowledge can be encoded as taxonomies of connection types, provided that connections can be represented as first-class entities within the ADL. For example, we found the need to specify classes of multicast groups and RMI connections in order to represent systems that dynamically "plug-and-play" with network components, and to simulate transient failures, transmission delays, and other network characteristics. Using connection types allowed us to more easily specify restrictions on events that pass among components, and to define constraints on inter-component behavior, while associating them directly with appropriate places in an architectural model. Making connections first class permits still further semantic distinction between components and connections, thus facilitating clear and explicit description of architecture. First class connections also encourage designers to define constraints for specific connection types and type hierarchies, so that formal reasoning about connector behavior can be localized. We suspect this may be of particular importance for architectures of dynamic systems, where connections provide a focal point for analysis.

5.3.3 Improvement to representation of behavior. As an adjunct to sending and receiving events, a Rapide component encapsulates a set of state variables. To test consistency conditions during execution, we needed to capture and analyze state variables maintained by multiple components. This required us to adopt several cumbersome solutions. We believe modeling of architectures for dynamic systems is greatly facilitated by permitting definition of component state from a subset of internal state variables. Component states should be selectively exported and recorded along with events. Linking events to changes in state [13] then allows recording of dependencies for analysis.

Assuming appropriate state variables are exported, further investigation is needed to determine how best to define, implement, and evaluate consistency conditions that involve the state of two or more components *and* that account for time. ADL constraint representation should include rich semantics for this purpose. Further, analyses of architectures for dynamic systems benefit greatly when ADL run-time environments include support for automated evaluation of inter-component consistency conditions (as some already do), and especially constraint languages and related constraint-analysis engines that account for time.

6. Conclusions and Future Work

Our current work illustrates the viability of an architecture-based approach to investigate and evaluate logical and behavioral properties of discovery protocols under conditions of dynamic change. Our results show that executable architecture models prove essential to understand the collective behavior of distributed systems. In this paper, we demonstrated how such models help to uncover ambiguity, incompleteness, and other issues in static, natural-language specifications. Our demonstration contributes to improving the specification for Jini. We also argue that a single executable architecture model can be used to investigate system performance as well as logical properties. Beyond this, we offered some recommendations, based on our experience, to improve the suitability of ADLs to model and analyze distributed systems.

In the next phase of our project, we intend to demonstrate that using architectural models provides a sound basis on which to compare and contrast the technical merits of various discovery protocols. The results from our analyses should provide industry with better understanding of the design and behavior of discovery protocols. We will define a generic set of usage scenarios to measure interesting events common among all protocols. These scenarios will exploit a common vocabulary and set of protocol features derived from our UML model. Similarly, we will identify a set of consistency conditions, design issues, and performance metrics that provide a suitable basis for comparison among discovery protocols.

We suspect relevant consistency conditions and metrics will involve only SMs and SUs, because not all protocols require SCMs.

The next phase of the project will also provide a vehicle for continued appraisal of our architecture-based approach to investigate distributed system designs under dynamic conditions. We intend to sharpen and refine our current assessment. We also hope to make more specific recommendations on ADL features to better support domain-specific models, to represent connections, and to analyze internal state of components. Modeling additional discovery protocols also provides an opportunity to examine reuse of architectural components as we attempt to adapt common functions in architectures for different protocols. This work should reveal insights regarding ADL features that facilitate reuse.

Finally, we suspect, but cannot yet conclude, that the nature of dynamism in the service-discovery domain differs from other real-time domains. The next phase of the project, together with the results of concurrent research in dynamic change within the defense software research community, should illuminate this issue as well. Since automatic component discovery and collaborative composition will be essential capabilities of future defense systems, early insights gained into this issue will likely prove important.

8. Acknowledgments

The work described in this paper benefits from financial support provided by the National Institute of Standards and Technology (NIST), the Advanced Research Development Agency (ARDA), and the Defense Advanced Research Projects Agency (DARPA). In particular, we acknowledge the support of Greg Puffenbarger from ARDA and John Salasin, DARPA's program manager for Dynamic Assembly for System Adaptability, Dependability, and Assurance (DASADA). We also gratefully acknowledge the insights that Jim Waldo, Jini Architect, provided us during several hours of discussion about our approach and preliminary results, and in written comments on an earlier draft of this paper.

9. References

[1] G. Bieber and J. Carpenter, "Openwings A Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems," on the www.openwings.org web site.
[2] Salutation Architecture Specification, Version 2.0c, Salutation Consortium, June 1, 1999.
[3] Universal Plug and Play Device Architecture, Version 1.0, Microsoft, June 8, 2000.
[4] Ken Arnold et al, The Jini Specification, V1.0 Addison-Wesley 1999. Latest version is 1.1 available from Sun.
[5] Specification of the Home Audio/Video Interoperability (HAVi) Archiecture, V1.1, HAVi, Inc., May 15, 2001.
[6] Service Location Protocol Version 2, Internet Engineering Task Force (IETF), RFC 2608, June 1999.
[7] Specification of the Bluetooth System, Core, Volume 1, Version 1.1, the Bluetooth SIG, Inc., February 22, 2001.
[8] B. Miller and R. Pascoe, Mapping Salutation Architecture APIs to Bluetooth Service Discovery Layer, Version 1.0, Bluetooth SIG White paper, July 1, 1999.
[9] C. Bettstetter and C. Renner, "A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol", *Proceedings of the Sixth EUNICE Open European Summer School: Innovative Internet Applications*, EUNICE 2000, Twente, Netherlands, September, 13-15, 2000.
[10] G. Richard, "Service Advertisement and Discovery: Enabling Universal Device Cooperation," *IEEE Internet Computing*, September-October 2000, pp. 18-26.
[11] B. Pascoe, "Salutation Architectures and the newly defined service discovery protocols from Microsoft and Sun: How does the Salutation Architecture stack up," Salutation Consortium whitepaper, June 6, 1999.
[12] Luckham, D. "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events," http://anna.stanford.edu/rapide, August 1996.
[13] Allen, R. "A Formal Approach to Software Architecture", Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997.
[14] Garlan, D, Monroe, R., and Wile, D., "Acme: An Architecture Description Interchange Language", Proceedings of CASCON '97, Nov. 1997
[15] Melton, R. "The Aesop System: A Tutorial," Carnegie Mellon University, Pittsburgh, Pennsylvania, 1998.
[16] Moriconi, M & Riemenschneider, R. "Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies," TR SRI-CSL-97-01, March 1997
[17] Medvidovic, N., P.Oreizy, J. Robbins, and R. Taylor. "Using Object-Oriented Typing to Support Architectural Design in the C2 Style", *Proceedings of SIGSOFT'96: The Fourth Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 16-18, 1996.
[18] Shaw, M. R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Eng*., vol. 21, no. 4, pp. 314-335, Apr. 1995.
[19] Vestal, S. MetaH User's Manual, Version 1.27, Honeywell Technology Center, Minneapolis, MN 55418, 1997.
[20] Shaw, M., R. DeLine, and G. Zelesnik, "Abstractions and Implementations for Architectural Connections," *Proc. Third Int'l Conf. Configurable Distributed Systems*, May 1996.
[21] Allen, R. and D. Garlan. "Formalizing Architectural Connection", *Proceedings of the Sixteenth International Conference on Software Engineering*, Sorrento, Italy, Ma 1994, pp. 71-80.
[22] J. Rekesh, UPnP, Jini and Salutation - A look at some popular coordination framework for future network devices, Technical Report, California Software Lab, 1999. Available online from http://www.cswl.com/whiteppr/tech/upnp.html.
[23] Garlan, D. "Higher Order Connectors", Workshop on Compositional Software Architectures, Monterey, CA, January, 1998.
[24] R. Pascoe, "Building Networks on the Fly", *IEEE Spectrum*, March 2001, pp. 61-65.